🚀 JavaScript Functions

(Detailed notes yet beginner friendly)

Why Functions Exists ?

Why Functions exist in JavaScript or in Programming in general ?

Start from the most basic problem

Imagine you are programming without functions.

Now :

- You have to write same code again and again
- Your code is NOT organized as complexity increases
- Code is NOT readable and maintainable.

Problem

Let's take an example of adding two numbers:

```
let a = 20;
let b = 30;
let c = a + b;
```

Now imagnine, you have many hundreds of variables to add, now if you write whole code again and again for each variable, it's NOT reusable and looks bad.

Solution: Use Functions

Write a addNumber function

```
addNumber(num1, num2){
return num1+num2
}
```

Why Functions Exists ?

Now in this function we can pass any values as an argument, many times, therefore we don't have to write code of 'adding two numbers' again and again

```
addNumber(25, 30)
addNumber(65,897)
addNumber (658, 28)
```

Now functions solves this problem.

What is Function ?

First Thought: A function is a reusable "code machine" or a "recipe."

Imagine you have a task you need to do over and over, like making a sandwich. Instead of re-writing the instructions (get bread, get cheese, etc.) every single time, you write the recipe down once and give it a name, like "Make Turkey Sandwich."

- The **recipe** is the function.
- The **ingredients** you give it (like "turkey" or "ham") are the parameters.
-
- The finished sandwich is the **return value**.

Why do we need them? The **DRY Principle**: Don't Repeat Yourself. Functions let you write a piece of logic once and reuse it hundreds of times.

A. The Three Core Parts

- **Declaration (Writing the Recipe)** : This is where you define the function and give it a name.
- **Calling (Using the Recipe)** : This is where you execute the function.
- **Parameters & Return Value (Ingredients & Finished Dish)** : How you pass data into a function and get a result out of it.

What is Function ?

B. Basic Syntax

```
// 1. DECLARATION: Writing the recipe
//    - `function` is the keyword.
//    - `greet` is the name of our function.
//    - `(name)` is the parameter, a placeholder for the input.

function greet(name) {
  // The code inside the {} is the "body" of the function.
  const message = `Hello, ${name}!`;
  return message; // 3. RETURN VALUE: Giving back the result.
}

// 2. CALLING: Using the recipe with a specific ingredient ("Alice")
const greetingForAlice = greet("Alice");
console.log(greetingForAlice); // Outputs: "Hello, Alice!"

const greetingForBob = greet("Bob");
console.log(greetingForBob);   // Outputs: "Hello, Bob!"
```

Parameter : The variable name inside the function's parentheses (name). It's a placeholder.
Argument : The actual value you pass into the function when you call it ("Alice").
Return Value : The value a function sends back using the return keyword. If a function has no return statement, it automatically returns undefined.

Different Ways to Create Functions

There are three main ways to create functions in JavaScript, and their differences are important.

A. Function Declaration

This is the classic way to create a function..

```
function add(a, b) {
  return a + b;
}
```

Key Feature : **Hoisting**. Function declarations are "hoisted," meaning the JavaScript engine "lifts" them to the top of their scope before the code is executed. This means you can call a function before you define it in the code.

```
console.log(add(2, 3)); // 5 (This works!)

function add(a, b) {
  return a + b;
}
```

Different Ways to Create Functions

B. Function Expression

Here, you create an anonymous (unnamed) function and assign it to a variable.

```
const subtract = function(a, b) {
  return a - b;
};
```

Key Feature : Not Hoisted. A function expression is not hoisted. It behaves just like a variable. You cannot call it before it is defined.

```
// subtract(10, 5); // This would throw a ReferenceError!

const subtract = function(a, b) {
  return a - b;
};

console.log(subtract(10, 5)); // 5 (This works)
```

C. Arrow Functions (ES6 - The Modern Way)

First Thought : "A shorter, cleaner syntax for writing functions."

Arrow functions are a more concise way to write function expressions. They are extremely popular in modern JavaScript.

```
// A function expression
const multiply = function(a, b) {
  return a * b;
}

// The same function as an arrow function
const multiplyArrow = (a, b) => {
  return a * b;
}
```

Syntax Rules for Arrow Functions:

1. Implicit Return : If the function body is just a single expression, you can remove the curly braces {} and the return keyword.

```
const multiplyShort = (a, b) => a * b; // The result of a * b
is automatically returned.
```

Different Ways to Create Functions

2. Single Parameter : If there is only one parameter, you can remove the parentheses ().

```
const square = x => x * x;
```

Advanced Parameter Concepts

A. Default Parameters (ES6)

You can provide a default value for a parameter in case it's not passed in (i.e., it's undefined).

```
// The default value for 'name' is "Guest".
function greet(name = "Guest") {
  console.log(`Hello, ${name}!`);
}

greet("Alice"); // Hello, Alice!
greet();        // Hello, Guest!
```

A. Default Parameters (ES6)

This allows a function to accept an indefinite number of arguments and gather them into a true array.

First Thought: "Put the rest of the ingredients into a single bag."

Advanced Parameter Concepts

```
// The '...numbers' gathers all arguments passed into an
array called 'numbers'.
function sumAll(...numbers) {
  let total = 0;
  for (const num of numbers) {
    total += num;
  }
  return total;
}

console.log(sumAll(1, 2));          // 3
console.log(sumAll(1, 2, 3, 4, 5));    // 15
console.log(sumAll(10));            // 10
```

The rest parameter must be the last parameter in the function definition.

A. Default Parameters (ES6)

This allows a function to accept an indefinite number of arguments and gather them into a true array.

First Thought: "Put the rest of the ingredients into a single bag."

Scope in Functions

First Thought: "A function is its own little world."

- **Global Scope** : Variables declared outside of any function are "global" and can be accessed from anywhere.
- **Function Scope (Local Scope)** : Variables declared inside a function (with **let**, **const**, or **var**) only exist inside that function. They cannot be accessed from the outside.

```
let globalVar = "I am global";

function myFunction() {
  let localVar = "I am local";
  console.log(globalVar); // "I am global" (Can access global variables)
  console.log(localVar);  // "I am local"
}

myFunction();

// console.log(localVar); // ReferenceError: localVar is not defined
```

This is a critical feature that prevents variables from different parts of your code from interfering with each other.

Rest vs Spread Operator

Here's the mind-bending part: they use the exact same syntax (...) but do opposite things!

The Key Difference

Spread = Unpack/Expand (takes one thing and breaks it apart)
Rest = Pack/Collect (takes many things and combines them into one)

Think of it like a suitcase (Analogy):
Spread = Opening a suitcase and spreading clothes everywhere
Rest = Gathering scattered clothes and packing them into a suitcase

SPREAD Operator (Expanding/Unpacking)

Arrays

Takes an array or object and spreads its elements out.

```
const numbers = [1, 2, 3];

// Spread: Unpack the array into individual elements
console.log(...numbers);  // Output: 1 2 3 (three separate values)
console.log(numbers);     // Output: [1, 2, 3] (one array)

// Practical use: Combining arrays
const moreNumbers = [4, 5, 6];
const combined = [...numbers, ...moreNumbers];
console.log(combined);  // [1, 2, 3, 4, 5, 6]

// Practical use: Copying an array
const copy = [...numbers];

// Practical use: Passing array elements as function arguments
function add(a, b, c) {
   return a + b + c;
}
console.log(add(...numbers));  // Same as add(1, 2, 3)
```

Rest vs Spread Operator

Objects

```
const person = { name: "Alice", age: 25 };

// Spread: Unpack object properties
const updatedPerson = { ...person, city: "NYC" };
console.log(updatedPerson);
// { name: "Alice", age: 25, city: "NYC" }

// Overwriting properties
const olderPerson = { ...person, age: 30 };
// { name: "Alice", age: 30 }
```

REST Operator (Collecting/Packing)

Rest vs Spread Operator

Objects

```
const person = { name: "Alice", age: 25 };

// Spread: Unpack object properties
const updatedPerson = { ...person, city: "NYC" };
console.log(updatedPerson);
// { name: "Alice", age: 25, city: "NYC" }

// Overwriting properties
const olderPerson = { ...person, age: 30 };
// { name: "Alice", age: 30 }
```

REST Operator (Collecting/Packing)

Takes **multiple individual elements** and **collects them into an array**.

In Function Parameters

```
// Rest: Collect all remaining arguments into an array
function sum(...numbers) {
    console.log(numbers);  // numbers is an array
    return numbers.reduce((total, num) => total + num, 0);
}

console.log(sum(1, 2, 3, 4, 5));  // 15
// Inside the function, numbers = [1, 2, 3, 4, 5]

// Mixing regular params with rest
function introduce(greeting, ...names) {
    console.log(greeting);  // "Hello"
    console.log(names);     // ["Alice", "Bob", "Charlie"]
}

introduce("Hello", "Alice", "Bob", "Charlie");
```

Rest vs Spread Operator

In Destructuring (Arrays)

```
const numbers = [1, 2, 3, 4, 5];

// Rest: Collect the "rest" of the elements
const [first, second, ...others] = numbers;

console.log(first);   // 1
console.log(second);  // 2
console.log(others);  // [3, 4, 5] - collected into an array!
```

In Destructuring (Objects)

```
const person = {
    name: "Alice",
    age: 25,
    city: "NYC",
    country: "USA"
};

// Rest: Collect remaining properties
const { name, ...otherInfo } = person;

console.log(name);       // "Alice"
console.log(otherInfo);  // { age: 25, city: "NYC", country: "USA" }
```

Rest vs Spread Operator

Side-by-Side Comparison (Rest & Spread Operator)

```
// SPREAD: Taking an array and spreading it out
const arr = [1, 2, 3];
console.log(...arr);  // 1 2 3 (three separate values)

// REST: Taking separate values and collecting them
function example(...params) {
   console.log(params);  // [1, 2, 3] (one array)
}
example(1, 2, 3);
```

How to Remember Which is Which

Look at WHERE it's used:

1. **RIGHT side of assignment = SPREAD (expanding)**

```
const copy = [...original];  // Spreading original
```

2. **LEFT side of assignment = REST (collecting)**

```
const [first, ...rest] = array;  // Collecting rest
```

3. **Function call = SPREAD (unpacking arguments)**

```
myFunction(...array);  // Spreading array into arguments
```

Rest vs Spread Operator

4. Function definition = REST (collecting parameters)

```
function myFunction(...params) { } // Collecting params
```

Real-World Example: Both Together!

```
function logDetails(name, age, ...hobbies) {
    // REST: hobbies collects remaining arguments into array
    console.log(`${name} is ${age} years old`);
    console.log("Hobbies:", hobbies);
}

const person = ["Alice", 25, "reading", "coding", "hiking"];

// SPREAD: Unpack array into individual arguments
logDetails(...person);

// Output:
// Alice is 25 years old
// Hobbies: ["reading", "coding", "hiking"]
```

QUICK RULE!

- **Spread** = ... takes ONE thing → makes it MANY
- **Rest** = ... takes MANY things → makes it ONE

Arrow Functions

1. The Core Purpose: A Better function

First Thought : Arrow functions were created to solve two major problems with traditional function expressions:

1. They were too verbose for simple, one-line operations (like in array methods).
2. Their behavior with the **this** keyword was a constant source of confusion and bugs.

An arrow function is a **syntactically compact alternative to a regular function expression.**

2. Syntax Breakdown: From Verbose to Concise

Let's transform a traditional function expression into its shortest possible arrow function form.

Start: Traditional Function Expression

```
const add = function(a, b) {
  return a + b;
};
```

Arrow Functions

Step 1: Replace function with =>

Remove the **function** keyword and place a "fat arrow" **=>** after the parameters.

```
const add = (a, b) => {
  return a + b;
};
```

Step 2: Implicit Return

This is the biggest syntax win. If the function body consists of only a **single return expression**, you can remove the curly braces **{}** and the **return** keyword. The result of the expression is returned automatically.

```
const add = (a, b) => a + b;
```

Step 3: Parentheses for a Single Parameter

If the function has **exactly one parameter**, you can also remove the parentheses around it.

```
// Traditional
const square = function(x) {
  return x * x;
};

// Arrow function with one parameter
const squareArrow = x => x * x;
```

Arrow Functions

If you have zero or more than one parameter, the parentheses are required.

```
const sayHello = () => "Hello"; // Zero parameters
const sum = (a, b, c) => a + b + c; // Three parameters
```

Important RULE: Returning an Object Literal

If you want to implicitly return an object literal, you must wrap it in parentheses to distinguish it from a function body's curly braces.

```
// WRONG: This is interpreted as a function body with a label, returning undefined.
const createUser = name => { name: name };

// CORRECT: Wrap the object in parentheses.
const createUserCorrect = name => ({ name: name });
```

Callback Function

A callback function in JavaScript is a function passed as an argument to another function. This allows the receiving function, often called a higher-order function, to execute the callback function at a specific point during its operation, typically after a particular task is completed.

The Analogy: Ordering a Pizza

Let's break this down.

The Scenario : You want a pizza, but it's going to take a while to cook. You don't want to just stand at the counter and wait. You want to go do something else.

1. **You place your order (You call a function)**
   You go to the pizza place and say, "I'd like a large pepperoni pizza." This is you calling a function, **orderPizza()**.

2. Y**ou provide the callback (You give them your phone number)**.
   The cashier says, "Great, that will be 20 minutes. What's your number so we can call you when it's ready?" That phone number is your callback function. It's the instruction you are leaving behind for what to do after the main task (making the pizza) is complete

   Your "phone number" function might be called **pickUpPizza**.

3. T**he long task happens (The pizza is made).**
   You leave and go about your business. The pizza place is busy making your pizza. This is the asynchronous part—a task is happening in the background, and your main program isn't blocked waiting for it.

Callback Function

2. **The task finishes, and they "call you back."**
   After 20 minutes, the pizza is ready. The cashier looks at your order, finds your phone number (**pickUpPizza**), and calls it.
   This is the callback execution.

The Code Equivalent

```
// This is your callback function. It's the "phone number" you'll leave behind.
// It describes what to do once the pizza is ready.
function pickUpPizza() {
  console.log("Pizza is ready! Driving to the store to pick it up.");
}


function orderPizza(callback) {


  console.log("Placing the pizza order...");

    console.log("Pizza is cooked!");

    callback();


}

// --- Let's run the program ---
// We call orderPizza and give it our pickUpPizza function as the callback argument.
orderPizza(pickUpPizza);

// This line will run immediately, while the pizza is still "cooking".
console.log("I'm not waiting at the store. I'm at home, coding.");
```

Output of the Code

```
Placing the pizza order...
I'm not waiting at the store. I'm at home, coding.
(after 2 seconds)
Pizza is cooked!
Pizza is ready! Driving to the store to pick it up.
```